Space-Profiling Semantics of the Call-by-Value Lambda Calculus and the CPS Transformation

Yasuhiko Minamide

Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba 305-8573, Japan

Abstract

We show that the CPS transformation from the call-by-value lambda calculus to a CPS language preserves space required for execution of a program within a constant factor. For the call-by-value lambda calculus we adopt a space-profiling semantics based on the profiling semantics of NESL by Blelloch and Greiner. However, we have noticed their semantics has some inconsistency between the treatments of stack space and heap space. This requires us to revise the semantics so that the semantics treats space in more consistent manner in order to obtain our result.

1 Introduction

In the studies of program transformations it is indispensable to prove their correctness. However, so far it is not so common to discuss effects on performance formally; it is usually discussed only by benchmarks. This sometimes results in undesirable situation. Several researchers have noticed that some program transformations are not safe with respect to space [1,3,10]. This clarifies that space safety of program transformations should be discussed formally.

To discuss space safety formally, we first need a space-profiling semantics which formalizes how much memory a program requires for its execution. However, it is not so easy to formulate such a semantics. Blelloch and Greiner proposed a space-profiling semantics for the parallel programming language NESL and discussed the space efficiency of an implementation of NESL [2]. As far as we know, this is the only work that formalized a space-profiling semantics of a programming language and discussed this topic formally. Their profiling semantics is an extension of the standard natural semantics of the call-by-value lambda calculus and seems suitable to study space safety of program transformations on call-by-value functional languages. However, the problem of formalizing space-profiling semantics is not completely solved. Their semantics is quite involved and it is not so clear that it models space required for execution of a program properly.

In this paper we show that the CPS transformation from the call-by-value lambda calculus to a CPS language preserves space within a constant factor by adopting a space-profiling semantics of Blelloch and Greiner for the lambda calculus. This result itself is significant since it confirms what we expected on space for the CPS transformation. Furthermore, this implies that the call-by-value lambda calculus has an implementation based on the CPS transformation where space is preserved and thus the space-profiling semantics of the lambda calculus models space required for execution of programs properly. However, during this study we have noticed that the semantics given by Blelloch and Greiner has some inconsistency between the treatments of stack space and heap space. This requires us to revise the semantics so that the semantics treats space in more consistent manner in order to obtain our result.

The CPS language we consider is basically a simplified version of the language used in the study of SML/NJ [1]. The language does not require stack to model execution of programs. This simplifies its space-profiling semantics. Furthermore, since the language is close to low level implementation we are relatively sure that the space-profiling semantics we define for the language models actual implementation. This is our motivation that we used the CPS language as the target language into which the call-by-value lambda calculus is translated.

This paper is organized as follows. We start by introducing the concept of profiling semantics and discussing what properties a program transformation should satisfy. As the property we discuss mainly in this paper we define *space efficient* program transformations. In Section 3 we introduce the profiling semantics of the call-by-value lambda calculus given by Blelloch and Greiner and discuss inconsistency between the treatments of stack space and heap space. In Section 4 the CPS language and its space-profiling semantics is introduced. In Section 5 we show that the CPS transformation is space efficient. We also consider some variations of profiling semantics of the call-by-value lambda calculus and the CPS language in Section 6. Finally we review related works and give directions for future work.

2 Profiling semantics, space efficiency, and space safety

In this section we review what is profiling semantics and what properties program transformations should satisfy on space. For simple functional languages the semantics can be defined as a function $eval(M)$ returning the result of evaluation if it terminates:

$$eval(M) = v$$

if the evaluation terminates and the result is the value v. Otherwise, $eval(M)$ is undefined. This semantics formalizes what should be the result of the computation of M, but does not give any information on how much time or space the evaluation requires.

Profiling semantics is introduced to discuss such properties formally. As an extension of the standard semantics, profiling semantics can be given as a function $eval(M)$ below.

$$eval(M) = (v, n)$$

This function gives the result of computation v and information n on resource requirement. In this paper we focus on space as the resource. More specially, we discuss not the space allocated totally, but the space required for execution in the presence of garbage collection. As the values of n we use non-negative integers that represent the size of space required for execution.

The next step is to formalize what property a program transformation should satisfy on space. Let us consider that a program M is translated to a program M' by a program transformation and the profiling semantics for the language after the transformation is given by $eval'(M')$. We say that a program transformation is *space efficient* if the following property holds.

There exist constants k_1 and k_2 such that for any program M

$$eval(M) = (v, n) \text{ implies } eval'(M') = (v', n') \text{ for some } n' \leq k_1 n + k_2.$$

In this definition we admit constants k_1 and k_2 because they seem dependent on the details of definition and not essential. It is clear that this condition ensures that the program transformation does not raise the space complexity of a program. This is the property shown by Blelloch and Greiner for an implementation of NESL [2] and we show this property for the CPS transformation.

However, this property might be too strong for some program transformations used in compilers. Appel discussed that a transformation is *space safe* if it does not raise the space complexity of programs [1]. To formalize this idea we have to consider programs with an input; we consider programs with one free variable and use the free variable for input. One way of formalizing space-safety is as follows:

For any program M with a free variable x, there exist constants k_1 and k_2 such that for any integer i the following holds:

$$eval(M[i/x]) = (v, n) \text{ implies } eval'(M'[i/x]) = (v', n')$$
$$\text{for some } n' \leq k_1 n + k_2$$

The key difference from space-efficiency is that the constants k_1 and k_2 are program-dependent. Minamide and Garrigue showed that this property with respet to time complexity for a type-directed unboxing transformation, which is not efficient in terms of the definition above [10]. We should remark that if we ignore inputs, the statement of space-safety gets trivial. The following claim is trivial on requirement of space.

For any program M, there exist program-dependent constants k_1 and k_2 such *that the following holds:*

$$eval(M) = (v, n) \textit{ implies } eval'(M') = (v', n') \textit{ for some } n' \leq k_1 n + k_2$$

It might be sufficient that a program transformation used in compilers is space safe. However, we think that it is interesting to show that a program transformation is space efficient, since it gives us a uniform constant independent of programs.

3 Lambda calculus and space-profiling semantics

In this section we introduce a space-profiling semantics of the call-by-value λ-calculus based on the space-profiling semantics of NESL by Blelloch and Greiner [2]. Their semantics does not explicitly take account of garbage collection, but the maximum size of reachable space during evaluation. We have noticed that this semantics has some inconsistency between the treatments of heap space and stack space: it takes account of the size of each closure, but ignores the size of each stack frame. This motivates us to revise the semantics so that the semantics treats space in more consistent manner. As we will see later, this revision is necessary to prove that the CPS transformation is space efficient. In this section we first describe the semantics by Blelloch and Greiner. Then we explain the problem and revise the semantics to make it more consistent.

We consider the following untyped call-by-value λ-calculus with a constant c.

$$M ::= x \mid c \mid \lambda x.M \mid M_1 M_2$$

The profiling operational semantics is given as an extension of the standard natural semantics. A value v is either a constant c or a location l. We use store σ mapping locations to store values sv to express sharing of values. A store value sv models a value allocated on heap memory and for this calculus it is only a closure. A closure $\langle \mathsf{cl}\ \rho, x, M\rangle$ consists of an environment ρ mapping variables to values, a variable, and an expression.

$$v ::= c \mid l$$

$$sv ::= \langle \mathsf{cl}\ \rho, x, M\rangle$$

We denote the set of the free variables of M by $FV(M)$ and for closures we write $FV(\langle \mathsf{cl}\ \rho, x, M\rangle)$ for $FV(M) \setminus \{x\}$.

The operational semantics is defined as a natural semantics with judgments of the following form

$$\rho, \sigma, R \vdash M \longmapsto v, \sigma; s$$

where ρ is environment, σ is store, R is a root set, and s is the maximum size of reachable space. A root set is a set of locations which is necessary after this computation.

Before giving the rules of the natural semantics we have to define reachable locations and the space of a set of locations. The set of locations $\mathit{locs}(l,\sigma)$ reachable from a location l in σ is defined as follows [1].

$$\mathit{locs}(l,\sigma) = \{l\} \cup \mathit{locs}(\sigma(l),\sigma)$$
$$\mathit{locs}(\langle \mathsf{cl}\ \rho, x, M\rangle, \sigma) = \textstyle\bigcup_{l\in L} \mathit{locs}(l,\sigma) \quad \text{where } L = \rho(FV(\langle \mathsf{cl}\ \rho, x, M\rangle))\lfloor_{\mathsf{Loc}}$$

where $S\lfloor_{\mathsf{Loc}}$ is the set of locations included in a set of values S.

The size of space $\mathit{space}(R,\sigma)$ reachable from a set of locations R is defined as follows:

$$\mathit{space}(R,\sigma) = \Sigma_{l\in \mathit{locs}(R,\sigma)} \mathit{size}(\sigma(l))$$
$$\mathit{size}(\langle \mathsf{cl}\ \rho, x, M\rangle) = |FV(\langle \mathsf{cl}\ \rho, x, M\rangle)| + 1$$

The size of a closure is proportional to the size of the environment required for M. This models the implementation of environments based on a flat record.

The following are the rules of the operational semantics, which are basically obtained by simplifying those given by Blelloch and Greiner.

(con)

$$\rho, \sigma, R \vdash c \longmapsto c, \sigma; \mathit{space}(R,\sigma)$$

(var)

$$\rho, \sigma, R \vdash x \longmapsto \rho(x), \sigma; \mathit{space}(R \cup \{\rho(x)\}\lfloor_{\mathsf{Loc}}, \sigma)$$

(abs)

$$\rho, \sigma, R \vdash \lambda x.M \longmapsto l, \sigma'; \mathit{space}(R \cup \{l\}, \sigma')$$

where l is a fresh location and $\sigma' = \sigma[\langle \mathsf{cl}\ \rho, x, M\rangle / l]$

(app)

$$\frac{\begin{array}{c}\rho, \sigma, R \cup \rho(FV(M_2)) \vdash M_1 \longmapsto l, \sigma_1; s_1 \\ \rho, \sigma_1, R \cup \{l\} \vdash M_2 \longmapsto v_2, \sigma_2; s_2 \\ \rho'[v_2/x], \sigma_2, R \vdash M_3 \longmapsto v, \sigma_3; s_3 \qquad \sigma_2(l) = \langle \mathsf{cl}\ \rho', x, M_3\rangle\end{array}}{\rho, \sigma, R \vdash M_1 M_2 \longmapsto v, \sigma_3; \mathit{max}(s_1+1, s_2+1, s_3)}$$

The size of reachable space is only calculated in the leaves of a derivation. In the rules (con), (var), and (abs) we have to take the root set R into account to determine the reachable space. Thus in (con) the size of reachable space is determined by $\mathit{space}(R,\sigma)$. In the rule of application we calculate the maximum size of reachable space by taking maximum of those of the subderivation. We calculate the maximum of s_1+1, s_2+1, and s_3 instead of s_1, s_2, and s_3 where adding 1 to s_1 and s_2 is explained as the space for stack. However, the evaluation of M_3 corresponds to a tail call. Thus we do not add 1 to s_3.

To define an evaluation function based on this natural semantics, we first define the observable values and the function $\mathit{oval}(v)$ coercing values into

[1] We assume that there is no cycle in a store. That is ensured for this operational semantics.

observable values as follows:

$$ov ::= c \mid \mathsf{cls}$$

$$oval(v) = \begin{cases} c & \text{if } v \text{ is a constant } c \\ \mathsf{cls} & \text{if } v \text{ is a location} \end{cases}$$

The space-profiling semantics based on the natural semantics is defined as follows:

$$eval_\lambda^{\mathsf{BG}}(M) = (ov, s) \textit{ if and only if } \emptyset, \emptyset, \emptyset \vdash M \longmapsto v, \sigma; s \textit{ and } ov = oval(v).$$

This seems reasonable definition, but if we think the evaluation more carefully, we will find some inconsistency in the rules of the natural semantics. The semantics takes account of the size of each closure, but it ignores the size of each stack frame. Let us consider the evaluation of application M_1M_2. During the evaluation of M_1, we have to preserve the values of $FV(M_2)$ in the stack frame, the size of which is not constant. However, this size is ignored in the rule of application.

We revise the rule of application so that the size of stack frames is treated in more consistent manner. The rule is revised as follows:

$$\frac{\begin{array}{c}\rho, \sigma, R \cup \rho(FV(M_2)) \vdash M_1 \longmapsto l, \sigma_1; s_1 \\ \rho, \sigma_1, R \cup \{l\} \vdash M_2 \longmapsto v_2, \sigma_2; s_2 \\ \rho'[v_2/x], \sigma_2, R \vdash M_3 \longmapsto v, \sigma_3; s_3 \qquad \sigma_2(l) = \langle \mathsf{cl}\, \rho', x, M_3 \rangle \end{array}}{\rho, \sigma, R \vdash M_1M_2 \longmapsto v, \sigma_3; max(s_1 + |FV(M_2)| + 1, s_2 + 1, s_3)}$$

We adopt this rule hereafter and refer the profiling semantics based on this rule $eval_\lambda(M)$. As we will see later, without this revision the CPS transformation is not space efficient.

These semantics $eval_\lambda^{\mathsf{BG}}(M)$ and $eval_\lambda(M)$ are not equivalent in terms of space efficiency of program transformations: there is no constants k_1 and k_2 such that

$$eval_\lambda^{BG}(M) = (ov, n) \textit{ implies } eval_\lambda(M) = (ov, n') \textit{ for some } n' \leq k_1 n + k_2.$$

This is verified by constructing expressions Z_n indexed by a natural number n where the space required for the execution of Z_n is proportional to n for $eval_\lambda^{\mathsf{BG}}(M)$ and n^2 for $eval_\lambda(M)$. The construction is given as follows:

$$\begin{aligned} X_n &= (\lambda x_1 \ldots x_n.c) z_1 \ldots z_n \\ Y_n &= (\lambda y_1 \ldots y_n.c) X_n \ldots X_n \\ Z_n &= (\lambda z_1 \ldots z_n.Y_n) c \ldots c \end{aligned}$$

where $\lambda x_1 \ldots x_n.M$ is abbreviation of $(\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M))))$. The intuition is that the evaluation of Z_n requires n stack frames of size n and thus n^2 space is necessary for $eval_\lambda(M)$, but only n space is necessary for $eval_\lambda^{\mathsf{BG}}(M)$ since the size of each stack frame is ignored.

4 CPS language and space-profiling semantics

As we saw in the previous section, it is delicate to formalize the space-profiling semantics even for the simple call-by-value λ-calculus. Thus in this section we consider a space-profiling semantics of a simpler CPS language where a program is expressed in continuation passing style. The language is basically a simplified version of the language used in the SML/NJ compiler [1]. The profiling semantics is defined based on the abstract machine proposed by Flanagan et al. [6] and does not require stack to model execution. That simplifies its space-profiling semantics.

The syntax of the CPS language is defined as follows:

$$V ::= x \mid c$$

$$M ::= x\langle V_1, \ldots, V_n\rangle \mid \texttt{let}\ x = \lambda x_1 \ldots x_n.M_1\ \texttt{in}\ M_2$$

A value expression V is either a variable x or a constant c. An expression M is either an application $x\langle V_1, \ldots, V_n\rangle$ with n-arguments $V_1, \ldots, V_n$ or a let-expression introducing a lambda abstraction. For readability we use $k, k_1, \ldots$ as variables of the CPS language for continuation.

We define the operational semantics based on the abstract machine proposed by Flanagan et al. [6] by extending it with locations and stores. The definition of store values, values, stores, and environments is similar to those for the semantics of the lambda calculus.

$$sv ::= \langle \mathsf{cl}\ \rho, x_1 \ldots x_n, M\rangle$$

$$v ::= l \mid c \mid \mathsf{stop}$$

The value stop is used to represent the initial continuation. We sometimes use an environment ρ as a function from value expressions to values by extending ρ with $\rho(c) = c$.

The operational semantics is defined as transition of a state Q consisting of a store, an environment, and an expression: $\langle \sigma, \rho, M\rangle$. The rules of the transition $\langle \sigma, \rho, M\rangle \longmapsto \langle \sigma', \rho', M'\rangle$ are defined as follows:

(app)

$$\langle \sigma, \rho, x\langle V_1, \ldots, V_n\rangle\rangle \longmapsto \langle \sigma, \rho'[\rho(V_1), \ldots, \rho(V_n)/z_1, \ldots, z_n], M\rangle$$

where $\sigma(\rho(x)) = \langle \mathsf{cl}\ \rho', z_1 \ldots z_n, M\rangle$.

(abs)

$$\langle \sigma, \rho, \texttt{let}\ x = \lambda x_1 \ldots x_j.M_1\ \texttt{in}\ M_2\rangle \longmapsto$$
$$\langle \sigma[\langle \mathsf{cl}\ \rho, x_1 \ldots x_j, M_1\rangle/l], \rho[l/x], M_2\rangle$$

where l is a fresh location.

As for the profiling operational semantics of the lambda calculus, we define the reachable locations and its space. The set of locations $locs(l, \sigma)$ reachable from l in σ is defined as follows.

$$\begin{aligned} locs(l,\sigma) &= \{l\} \cup locs(\sigma(l),\sigma) \\ locs(\langle \mathsf{cl}\, \rho, x_1 \dots x_n, M\rangle, \sigma) &= \textstyle\bigcup_{l\in L} locs(l,\sigma) \end{aligned}$$

where $L = \rho(FV(M) \setminus \{x_1,\dots,x_n\})|_{\mathsf{Loc}}$.

The size of space reachable from a set of locations R is defined as follows:

$$\begin{aligned} space(R,\sigma) &= \Sigma_{l\in locs(R,\sigma)} size(\sigma(l)) \\ size(\langle \mathsf{cl}\, \rho, x_1 \dots x_n, M\rangle) &= |FV(M) \setminus \{x_1,\dots,x_n\}| + 1 \end{aligned}$$

The set of locations and the size of space reachable from a state $\langle\sigma,\rho,M\rangle$ defined as follows:

$$\begin{aligned} locs(\langle\sigma,\rho,M\rangle) &= \textstyle\bigcup_{l\in L} locs(l,\sigma) \qquad \text{where } L = \rho(FV(M))|_{\mathsf{Loc}} \\ space(\langle\sigma,\rho,M\rangle) &= space(locs(\langle\sigma,\rho,M\rangle),\sigma) \end{aligned}$$

We write $Q_1 \longmapsto_s^* Q_n$ if $Q_1 \longmapsto^* Q_n$ and $s = \mathsf{max}_{1\le i\le n}(space(Q_i))$. Let M be a program with a variable k for the initial continuation. Then we define the profiling semantics of M as follows:

$$eval_{\mathsf{cps}}(M) = (ov, s)$$

if and only if

$$\langle\emptyset, [\mathsf{stop}/k], M\rangle \longmapsto_s^* \langle\sigma,\rho,k'\langle V\rangle\rangle$$

and $oval(\rho(V)) = ov$ and $\rho(k') = \mathsf{stop}$.

5 CPS transformation is space efficient

In this section we will consider a CPS transformation from the lambda calculus to the CPS language and show that the transformation is space efficient. The proof simultaneously shows that the transformation preserves the observable behavior of programs and is space efficient.

We define the CPS transformation as a deductive system with judgments of the form of $k \vdash M \rightsquigarrow M'$ which means M is transformed to M' by using k as a variable for continuation. The rules of the transformation are given as follows:

$$k \vdash x \rightsquigarrow k\langle x\rangle$$

$$k \vdash c \rightsquigarrow k\langle c\rangle$$

$$\frac{k \vdash M \rightsquigarrow M'}{k \vdash \lambda x.M \rightsquigarrow \texttt{let}\ f = \lambda xk.M'\ \texttt{in}\ k\langle f\rangle}$$

$$\frac{k_1 \vdash M_1 \rightsquigarrow M_1' \qquad k_2 \vdash M_2 \rightsquigarrow M_2'}{k \vdash M_1M_2 \rightsquigarrow \texttt{let}\ k_1 = \lambda f.\texttt{let}\ k_2 = \lambda a.f\langle ak\rangle\ \texttt{in}\ M_2'\ \texttt{in}\ M_1'}$$

We use the CPS language as the target language of the transformation instead of the lambda calculus and for readability we introduce fresh variables for continuation in the transformation of applications. This is basically the standard CPS transformation by Fischer and Plotkin [5,13].

The following lemma determines the set of the free variables of a CPS-transformed expression.

Lemma 5.1 *If $k \vdash M \rightsquigarrow M'$, then $FV(M') = FV(M) \cup \{k\}$.*

In the transformation of application, the continuation for M_1' is the following expression.

$$\lambda f.\mathtt{let}\ k_2 = \lambda a.f\langle ak\rangle\ \mathtt{in}\ M_2'$$

The set of free variables of the expression is $(FV(M_2')\backslash\{k_2\})\cup\{k\} = FV(M_2)\cup\{k\}$. Thus the size of the closure for this expression is not a constant, but $|FV(M_2)| + 2$. This is the reason why we need the revised space-profiling semantics of the lambda calculus to show that the CPS transformation is space efficient.

The following theorem claims that the CPS transformation is space efficient for the constant $K = 3$.

Theorem 5.2 *Let $k \vdash M \rightsquigarrow M'$. If $eval_\lambda(M) = (ov, s)$, then $eval_{\mathsf{cps}}(M') = (ov, s')$ and $s' \leq Ks$.*

The constant K is determined as follows. Let us consider the evaluation of M_1M_2 which is translated into the following expression.

$$\mathtt{let}\ k_1 = \lambda f.\mathtt{let}\ k_2 = \lambda a.f\langle ak\rangle\ \mathtt{in}\ M_2'\ \mathtt{in}\ M_1'$$

When we evaluate M_2', the current continuation is $\langle \mathsf{cl}\ \rho, a, f\langle ak\rangle\rangle$ for some environment ρ. The size of this closure is 3 since the set of the free variables is $\{f, k\}$. This size should correspond to the size of the stack frame to evaluate M_2 which is 1 in the rule for application in the definition of $eval_\lambda(M)$. Thus we adopt 3 for K.

In order to prove this theorem we have to generalize the claim of the theorem. First, in order to show that the CPS transformation preserves the observable behavior of programs, we define the relations $\approx$ between values, store values, environments, and stores of the lambda calculus and the CPS language as follows:

$$\begin{array}{rcll} & c \approx c & & \\ & l \approx l & & \\ \\ \langle \mathsf{cl}\ \rho, x, M\rangle & \approx & \langle \mathsf{cl}\ \rho', xk, M'\rangle & \text{if } \rho \approx \rho' \text{ and } k \vdash M \rightsquigarrow M' \\ & \rho \approx \rho' & & \text{if } \rho(x) \approx \rho'(x) \text{ for all } x \in Dom(\rho) \\ \\ & \sigma \approx \sigma' & & \text{if } \sigma(l) \approx \sigma'(l) \text{ for all } l \in Dom(\sigma) \end{array}$$

Since a store σ' of the CPS language includes locations pointing closures for continuation, we ignore the locations not included in the domain of σ of the lambda calculus in this definition.

The following lemma guarantees that for the related stores the set of reachable locations and the size of reachable space coincide.

Lemma 5.3 *Let $\sigma \approx \sigma'$ and R be a set of locations with $R \subseteq Dom(\sigma)$. Then $locs(R, \sigma) = locs(R, \sigma')$ and $space(R, \sigma) = space(R, \sigma')$.*

Secondly, we have to distinguish locations of the CPS language for continuation from those corresponding closures of the lambda calculus. Let σ and σ' be the stores of the lambda calculus and the CPS language respectively. The locations corresponding to continuation $klocs(v_k, \sigma, \sigma')$ and the locations corresponding to heap $hlocs(v_k, \sigma, \sigma')$ are defined as follows:

$$klocs(v_k, \sigma, \sigma') = locs(v_k, \sigma') \setminus Dom(\sigma)$$
$$hlocs(v_k, \sigma, \sigma') = locs(v_k, \sigma') \cap Dom(\sigma)$$

Then the size of space corresponding to continuation $kspace(v_k, \sigma, \sigma')$ is defined as follows:

$$kspace(v_k, \sigma, \sigma') = space(klocs(v_k, \sigma, \sigma'), \sigma')$$

The main theorem is generalized to the following lemma. The proof of the lemma appears in the appendix. The theorem is obtained by considering the empty stores for σ and σ', the empty environments for ρ and ρ', the empty set for R, and the initial continuation stop for v_k.

Lemma 5.4 *Let $k \vdash M \rightsquigarrow M'$, $\sigma \approx \sigma'$, $\rho \approx \rho'$, and $hlocs(v_k, \sigma, \sigma') = locs(R, \sigma)$.*
If $\rho, \sigma, R \vdash M \longmapsto v, \sigma_0; s$, then $\langle \sigma', \rho'[v_k/k], M' \rangle \rightarrow^*_{s'} \langle \sigma'_0, \rho''[v_k/k], k\langle V \rangle \rangle$ and $\sigma_0 \approx \sigma'_0$ and $v \approx \rho''(V)$ and $s' \leq Ks + kspace(v_k, \sigma, \sigma')$.

6 Variations of profiling semantics

In this section we consider some variations of profiling semantics of the call-by-value lambda calculus and the CPS language.

In the profiling semantics of the λ-calculus, we have not considered much about constant factor on the stack space. When we evaluate M_2 of application $M_1 M_2$, it is actually considered that the space of size 2 is necessary at least for the value l of M_1 and the return address. Then the rule is revised as follows:

$$\frac{\begin{array}{c} \rho, \sigma, R \cup \rho(FV(M_2)) \vdash M_1 \longmapsto l, \sigma_1; s_1 \\ \rho, \sigma_1, R \cup \{l\} \vdash M_2 \longmapsto v_2, \sigma_2; s_2 \\ \rho'[v_2/x], \sigma_2, R \vdash M_3 \longmapsto v, \sigma_3; s_3 \qquad \sigma_2(l) = \langle \mathsf{cl}\ \rho', x, M_3 \rangle \end{array}}{\rho, \sigma, R \vdash M_1 M_2 \longmapsto v, \sigma_3; max(s_1 + |FV(M_2)| + 1, s_2 + 2, s_3)}$$

If we adopt this rule, the constant K that is used to show that the CPS transformation is space efficient can be reduced to 2. In this sense it is considered that $K = 3$ is overestimated. However, the concrete value of this constant K will depend on the details of implementations anyway.

In this paper we have taken an approach to counting the size of closures and stack frames to make the semantics of the lambda calculus consistent. However, it is also possible to ignore the size of both closures and stack frames. Even if we take this approach, we believe that the CPS transformation is space efficient. However, this approach will make some transformation which is not space efficient if we take the size of closures and stack frames into account space efficient. That is against our intuition and we consider that this approach is undesirable.

In the space-profiling semantics of the CPS language, we have taken account of only the size of reachable locations as the size of a state and ignored the size of the environment ρ of the state. If we consider the size of the environment, the definition of the space of a state (ρ, σ, M) is revised as follows:

$$\mathit{space}'(\langle \rho, \sigma, M\rangle) = |FV(M)| + \mathit{space}(\mathit{locs}(\langle \rho, \sigma, M\rangle))$$

Here, we add $|FV(M)|$ instead of $|Dom(\rho)|$ to discard the space for the values that are not necessary to evaluate M. This definition $\mathit{space}'(Q)$ is also a reasonable definition and it is not so clear which definition we should take. The two semantics are not equivalent in the sense of space efficiency. However, if we restrict programs to those obtained by the CPS transformation, two semantics are equivalent. Let $\mathit{eval}'_{\mathsf{cps}}(M)$ be the profiling semantics defined based on this definition. If $k \vdash M \leadsto M'$, the following holds.

$$\textit{If } \mathit{eval}_{\mathsf{cps}}(M') = (v, n) \textit{ and } \mathit{eval}'_{\mathsf{cps}}(M') = (v, n'), \textit{ then } n' \leq 2n + 3.$$

This means the two profiling semantics are equivalent in the sense of space efficiency for the programs obtained by the CPS transformation.

7 Related work

We have concentrated on issues of space in this paper, but it is also important to discuss execution time formally. A semantics profiling time is relatively easy to formulate and several researchers have studied effect of program transformations on execution time [14,7,2]. Even for time some advanced program transformation was shown that it may raise time complexity of a program. Minamide and Garrigue showed that the type-directed unboxing transformation proposed by Leroy [9] has this problem. They proposed a refinement of this transformation and proved that their refinement preserves time complexity by the method of logical relations [10].

Several researchers have noticed that some program transformations are not safe with respect to space and proposed refinements of the transformations to avoid that problem [15,10]. Shao and Appel proposed a closure conversion with a sophisticated environment representation that is space safe [15]. Minamide and Garrigue expected that their refinement of unboxing transformation preserves space complexity with respect to heap space as well as time complexity [10]. However, no proof is given in both studies.

Blelloch and Greiner proposed two profiling semantics of NESL: one based on natural semantics and the other based on an abstract machine [2]. They showed that the implementation based on the latter semantics is time and space efficient with respect to the former semantics. However, as we discussed in this paper the treatment of stack space and heap space on both semantics seems inconsistent and different from ours. In another recent effort to address space-safety formally, Gustavsson and Sands has developed a theory of a space improvement relation for a call-by-need programming languages and showed space safety of inlining of affine-linear bindings [8].

Morrisett et al. proposed an operational semantics where various methods of garbage collection can be discussed formally [11,12]. In this paper we have ignored details of garbage collection and focused on the size of reachable space as Blelloch and Greiner did. It is desirable to show the connection of the size of reachable space and the space required for execution of a program formally.

8 Future work

We have considered the standard CPS transformation which introduces administrative redexes [13]. However, it might be more natural to consider an advanced CPS transformation as implementation of the call-by-value lambda calculus, where administrative redexes are eliminated at time of the CPS transformation. Thus we think that it is important to show such a CPS transformation is also space efficient. For such a CPS transformation, it is not straightforward to preserve tail calls [4]. By showing that such a CPS transformation is space efficient, we can indirectly show that tail calls are preserved.

The CPS language we considered is close to low level implementation. However, actual compilers still use other program transformations such as closure conversion to obtain executable code. To ensure that the profiling semantics of the call-by-value lambda calculus models execution properly we should prove that such transformations are also space efficient.

In this paper we have considered space efficiency of program transformations. However, there will be program transformations used in compilers which are not space efficient, but only space safe. It will be interesting to clarify which program transformations are not space efficient, but space safe.

Acknowledgement

We would like to thank Manuel Chakravarty and the anonymous reviewers for their many helpful comments. This work is partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Encouragement of Young Scientists of Japan, No. 11780216, 1999.

References

[1] A. W. Appel. *Compiling with Continuation.* Cambridge University Press, 1992.

[2] G. E. Blelloch and J. Greiner. A provably time and space efficient implementation of NESL. In *Proc. of ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, 1996.

[3] D. R. Chase. Safety consideration for storage allocation optimization. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–10, 1988.

[4] O. Danvy and A. Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361 – 391, 1992.

[5] M. Fischer. Lambda calculus schemata. In *Proc. of the ACM Conference on Proving Assertions About Programs, SIGPLAN Notices,* 7(1), 1972.

[6] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

[7] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. In *Proc. of ACM Symposium on Principles of Programming Languages*, pages 309 – 321, 1996.

[8] J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In *Proc. of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS99)*, 1999.

[9] X. Leroy. Unboxed objects and polymorphic typing. In *Proc. of ACM Symposium on Principles of Programming Languages*, pages 177 – 188, 1992.

[10] Y. Minamide and J. Garrigue. On the runtime complexity of type-directed unboxing. In *Proc. of ACM SIGPLAN International Conference on Functional Programming*, pages 1–12, 1998.

[11] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proc. of ACM Symposium on Functional Programming Languages and Computer Architecture*, pages 66–77, 1995.

[12] G. Morrisett and R. Harper. Semantics of memory management for polymorphic languages. Technical report, School of Computer Science, Carnegie Mellon University, 1996. CMU-CS-96-176.

[13] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[14] D. Sands. Complexity analysis for a lazy higher-order language. In *Proc. of European Symposium on Programming*, pages 361 – 375, 1990.

[15] Z. Shao and A. W. Appel. Space-efficient closure representations. In *Proc. of ACM Conference on Lisp and Functional Programming*, pages 150–161, 1994.

A Proof of Lemma 5.4.

Proof. By induction on the derivation of $\rho, \sigma, R \vdash M \longmapsto v, \sigma_0; s$. We only present the cases for applications and abstractions. The cases for the constant and variables are similar to the case of abstractions.

Case: $\rho, \sigma, R \vdash \lambda x.M \longmapsto l, \sigma_0; \mathit{space}(R \cup \{l\}, \sigma_0)$ where $\sigma_0 = \sigma[\langle \mathsf{cl}\, \rho, x, M\rangle/l]$. The expression is translated to $\texttt{let}\ f = \lambda xk.M'\ \texttt{in}\ k\langle f\rangle$. Then,

$$\langle \rho'[v_k/k], \sigma', \texttt{let}\ f = \lambda xk.M'\ \texttt{in}\ k\langle f\rangle\rangle \longmapsto$$
$$\langle \rho'[v_k/k][l/f], \sigma'[\langle \mathsf{cl}\ \rho', xk, M'\rangle/l], k\langle f\rangle\rangle$$

where $\sigma[\langle \mathsf{cl}\ \rho, x, M\rangle/l] \approx \sigma'[\langle \mathsf{cl}\ \rho', xk, M'\rangle/l]$ and $l \approx l$. Let us name the states above as follows:

$$Q_1 \equiv \langle \rho'[v_k/k], \sigma', \texttt{let}\ f = \lambda xk.M'\ \texttt{in}\ k\langle f\rangle\rangle$$
$$Q_2 \equiv \langle \rho'[v_k/k][l/f], \sigma'[\langle \mathsf{cl}\ \rho', xk, M'\rangle/l], k\langle f\rangle\rangle$$

We can calculate $\mathit{locs}(Q_2)$ as follows:

$$\begin{aligned} \mathit{locs}(Q_2) &= \mathit{klocs}(v_k, \sigma, \sigma') \cup \mathit{hlocs}(v_k, \sigma, \sigma') \cup \mathit{locs}(l, \sigma') \\ &= \mathit{klocs}(v_k, \sigma, \sigma') \cup (\mathit{locs}(R, \sigma) \cup \mathit{locs}(l, \sigma')) \end{aligned}$$

Hence

$$\mathit{space}(Q_2) \leq s + \mathit{kspace}(v_k, \sigma, \sigma')$$

This completes the proof of this case since $\mathit{space}(Q_1) \leq \mathit{space}(Q_2)$.

Case: $\rho, \sigma, R \vdash M_1 M_2 \longmapsto v, \sigma_0; s$ is derived from $\rho, \sigma, R_1 \vdash M_1 \longmapsto l, \sigma_1; s_1$ and $\rho, \sigma_1, R_2 \vdash M_2 \longmapsto v_2, \sigma_2; s_2$ and $\rho_3[v_2/x], \sigma_2, R \vdash M_3 \longmapsto v, \sigma_0; s_3$ where $\sigma_1(l) = \langle \mathsf{cl}\ \rho_3, x, M_3\rangle$ and $s = \mathit{max}(s_1 + |FV(M_2)| + 1, s_2 + 1, s_3)$.

The expression is translated to the following expression

$$\texttt{let}\ k_1 = \lambda f.\texttt{let}\ k_2 = \lambda a.f\langle ak\rangle\ \texttt{in}\ M_2'\ \texttt{in}\ M_1'$$

where $k_1 \vdash M_1 \rightsquigarrow M_1'$ and $k_2 \vdash M_2 \rightsquigarrow M_2'$. Let ρ_0' be $\rho'[v_k/k]$. We have the following transition.

$$\langle \rho_0', \sigma, \texttt{let}\ k_1 = \lambda f.\texttt{let}\ k_2 = \lambda a.f\langle ak\rangle\ \texttt{in}\ M_2'\ \texttt{in}\ M_1'\rangle \longmapsto$$
$$\langle \rho_0'[l_{k_1}/k_1], \sigma'[v_{k_1}/l_{k_1}], M_1'\rangle$$

where $v_{k_1} = \langle \mathsf{cl}\ \rho_0', f, \texttt{let}\ k_2 = \lambda a.f\langle ak\rangle\ \texttt{in}\ M_2'\rangle$ and $\sigma_1' = \sigma'[v_{k_1}/l_{k_1}]$.

Let $R_1 = R \cup \rho(FV(M_2))$. Then

$$\begin{aligned} \mathit{hlocs}(v_{k_1}, \sigma, \sigma_1') &= \mathit{hlocs}(v_k, \sigma, \sigma') \cup \mathit{locs}(\rho_0'(FV(M_2))) \\ &= R \cup \mathit{locs}(\rho(FV(M_2)), \sigma) = R_1 \end{aligned}$$

and $\rho \approx \rho_0'$. By induction hypothesis for M_1,

$$\langle \rho_0'[l_{k_1}/k_1], \sigma_1', M_1'\rangle \longmapsto^*_{s_1'} \langle \rho''[l_{k_1}/k_1], \sigma_1'', k\langle x\rangle\rangle$$

and $\sigma_1 \approx \sigma_1''$ and $l \approx \rho''(x)$ and $s_1' \leq K s_1 + \mathit{kspace}(v_{k_1}, \sigma, \sigma_1')$. Thus $\rho''(x) = l$ and $\sigma_1''(l)$ must be the form of $\langle \mathsf{cl}\ \rho_3', xk, M_3'\rangle$ and $k \vdash M_3 \rightsquigarrow M_3'$.

Let v'_1 be $\rho''(x)$.

$$\begin{aligned}\langle \rho''[v_{k_1}/k_1], \sigma''_1, k\langle x\rangle\rangle &\longmapsto_{s'_1} \langle \rho'_0[l/f], \sigma''_1, \mathtt{let}\ k_2 = \lambda a.f\langle ak\rangle\ \mathtt{in}\ M'_2\rangle \\ &\longmapsto \langle \rho'_0[l/f][l_{k_2}/k_2], \sigma''_1[v_{k_2}/l_{k_2}], M'_2\rangle\end{aligned}$$

where $v_{k_2} = \langle \mathsf{cl}\, \rho'[l/f], a, f\langle ak\rangle\rangle$.

Let $R_2 = R \cup \{l\}$. We have

$$hlocs(v_{k_2}, \sigma_2, \sigma'_2) = hlocs(v_k, \sigma, \sigma') \cup \{l\} = R_2$$

and $\rho \approx \rho'_0[l/f]$. By induction hypothesis on M_2,

$$\langle \rho'_0[l/f][v_{k_2}/k_2], \sigma''_1[v_{k_2}/l_{k_2}], M'_2\rangle \longmapsto^*_{s'_2} \langle \rho''_2[v_{k_2}/k_2], \sigma'_2, k\langle y\rangle\rangle$$

and $\sigma_2 \approx \sigma'_2$ and $v_2 \approx \rho''_2(y)$ and $s'_2 \leq Ks_2 + kspace(v_{k_2}, \sigma_2, \sigma'_2)$.

Let v'_2 be $\rho''_2(y)$.

$$\begin{aligned}\langle \rho''_2[v_{k_2}/k], \sigma'_2, k\langle y\rangle\rangle &\longmapsto \langle \rho'_0[l/f][v'_2/a], \sigma'_2, f\langle ak\rangle\rangle \\ &\longmapsto \langle \rho'_3[v'_2/x][v_k/k], \sigma'_2, M'_3\rangle\end{aligned}$$

Since $\rho_3[v_2/x] \approx \rho'_3[v'_2/x]$, by induction hypothesis on the evaluation of M_3

$$\langle \rho'_3[v'_2/x][v_k/k], \sigma'_2, M'_3\rangle \longmapsto^*_{s'_3} \langle \rho''_3[v_k/k], \sigma'_0, k\langle z\rangle\rangle$$

and $\sigma_0 \approx \sigma'_0$ and $v \approx \rho''_3(z)$ and $s'_3 \leq s_3 + klocs(v_k, \sigma_2, \sigma'_2)$.

By summarizing, we have the following transition.

$$\begin{array}{rc}
Q_0 = & \langle \rho'_0, \sigma, \mathtt{let}\ k_1 = \lambda f.\mathtt{let}\ k_2 = \lambda a.f\langle ak\rangle\ \mathtt{in}\ M'_2\ \mathtt{in}\ M'_1\rangle \\
 & \downarrow \\
Q'_0 = & \langle \rho'_0[l_{k_1}/k_1], \sigma'[v_{k_1}/l_{k_1}], M'_1\rangle \\
 & \downarrow^{s'_1}_{*} \\
Q_1 = & \langle \rho''[l_{k_1}/k_1], \sigma''_1, k\langle x\rangle\rangle \\
 & \downarrow \\
Q'_1 = & \langle \rho'_0[l/f], \sigma''_1, \mathtt{let}\ k_2 = \lambda a.f\langle ak\rangle\ \mathtt{in}\ M'_2\rangle \\
 & \downarrow \\
 & \langle \rho'_0[l/f][l_{k_2}/k_2], \sigma''_1[v_{k_2}/l_{k_2}], M'_2\rangle \\
 & \downarrow^{s'_2}_{*} \\
Q_2 = & \langle \rho''_2[v_{k_2}/k_2], \sigma'_2, k\langle y\rangle\rangle \\
 & \downarrow \\
Q'_2 = & \langle \rho'_0[l/f][v'_2/a], \sigma'_2, f\langle ak\rangle\rangle \\
 & \downarrow \\
 & \langle \rho'_3[v_2/a][v_k/k], \sigma'_2, M'_3\rangle \\
 & \downarrow^{s'_3}_{*} \\
 & \langle \rho''_3[v_k/k], \sigma'_0, k\langle z\rangle\rangle
\end{array}$$

By checking reachable locations from the states above, we obtain the following relations.

$$\begin{aligned}space(Q_0) \leq space(Q'_0) &\leq s'_1 \\ space(Q'_1) \leq space(Q_1) &\leq s'_1 \\ space(Q'_2) \leq space(Q_2) &\leq s'_2\end{aligned}$$

Hence we obtain

$$\langle \sigma', \rho'[v_k/k], M' \rangle \to^*_{s'} \langle \sigma'_0.\rho''[v_k/k], k\langle x \rangle \rangle$$

for $s' = max(s'_1, s'_2, s'_3)$.

Then the following calculation completes the proof of this case.

$$\begin{aligned}
s'_1 &\le Ks_1 + kspace(l_{k_1}, \sigma, \sigma'[v_{k_1}/l_{k_1}]) \\
&= Ks_1 + kspace(v_k, \sigma, \sigma') + |FV(M_2)| + 2 \\
&\le Ks + kspace(v_k, \sigma, \sigma') \\
s'_2 &\le Ks_2 + kspace(l_{k_2}, \sigma_1, \sigma''_1[v_{k_2}/l_{k_2}]) \\
&= Ks_2 + kspace(v_k, \sigma, \sigma') + 3 \\
&\le Ks + kspace(v_k, \sigma, \sigma') \\
s'_3 &\le Ks_3 + kspace(v_k, \sigma_2, \sigma'_2) \\
&\le Ks + kspace(v_k, \sigma, \sigma')
\end{aligned}$$

□